Comp 311
Functional Programming

Eric Allen, PhD
Vice President, Engineering
Two Sigma Investments, LLC
Comp 311 Homework 1 Hours Spent

Number of Students

Hours

0  5  10  15  20

0  5  10  15  20
Comp 311 Homework 1 More Time Needed

Number of Students

Hours

1 2 3 4 5 6
Comp 311 Homework 1 Workload

Number of Students

Rating
Comp 311 Homework 1 Enjoyable

![Bar Chart]

- **Number of Students**
- **Rating**

- 0 2 4 6 8 10
- 1 2 3 4 5

- The chart shows the number of students' ratings for Comp 311 Homework 1 Enjoyable.
Comp 311 Ease of Following Lectures

Number of Students

Rating
Actions

- Switch to two week assignments
- Double the weighting on subsequent assignments
- Keep Thursday at 2:30pm deadline
Importing a Member of a Package

import scala.collection.immutable.List
Importing Multiple Members of a Package

import scala.collection.immutable.{List, Vector}
Importing and Renaming Members of a Package

import scala.collection.immutable.{List=>SList, Vector}
Importing All Members of a Package

import scala.collection.immutable._

Note that * is a valid identifier in Scala!
Combining Notations

import scala.collection.immutable.{_}

same meaning as:

import scala.collection.immutable.immutable._
Combining Notations

import scala.collection.immutable.{List=>SList,_}

Imports all members of the package but renames List to SList
Combining Notations

```scala
import scala.collection.immutable.{List=>_,_}

Imports all members of the package except for List
```
Importing a Package

import scala.collection.immutable

Now sub-packages can be denoted by shorter names:

immutable.List
Importing and Renaming Packages

```scala
import scala.collection.{immutable => I}

Allows members to be written like this:

I.List
```
Importing Members of An Object

import Arithmetic._

Allows members such as `Arithmetic.gcd` to be write like this:

```
gcd
```
Implicit Imports

The following imports are implicitly included in your program:

```scala
import java.lang._
import scala._
import Predef._
```
Package java.lang

• Contains all the standard Java classes

• This import allows you to write things like:

    Thread

    instead of:

    java.lang.Thread
Package scala

- Provides access to the standard Scala classes:
  
  BigInt, BigDecimal, List, etc.
Object Predef

- Definitions of many commonly used types and methods, such as:

  require, ensuring, assert
Visibility Modifier Private

For a method `Arithmetic.reduce` in package `Rationals`

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>no modifier</td>
<td>public access</td>
</tr>
<tr>
<td>private</td>
<td>private to class Arithmetic</td>
</tr>
</tbody>
</table>
Local Definitions

• As with constant definitions, we can make function definitions local to the body of a function

• The functions can be referred to only in the body of the enclosing function
def reduce() = {
    val isPositive = 
        ((numerator < 0) & (denominator < 0)) | 
        ((numerator > 0) & (denominator > 0))

    def reduceFromInts(num: Int, denom: Int) = {
        require ((num >= 0) & (denom > 0))
        val gcd = Arithmetic.gcd(num, denom)
        val newNum = num/gcd
        val newDenom = denom/gcd

        if (isPositive) Rational(newNum, newDenom)
        else Rational(-newNum, newDenom)
    }
    reduceFromInts(Arithmetic.abs(numerator), Arithmetic.abs(denominator))
}

} ensuring (_ match {
    case Rational(n,d) => Arithmetic.gcd(n,d) == 1 & (d > 0)
})
Design Templates for Abstract Datatypes (Part 2)
Case Two
We Expect Many New Functions But Few New Variants
Case 2: We Expect Many New Functions But Few New Variants

- This is a case that traditional functional programming handles well

- Classic example domains: Compilers, theorem provers, numeric algorithms, machine learning

- Declare a top-level function with cases for each data variant
  a.k.a., The Visitor Pattern
Again We Turn to Pattern Matching

val pi = 3.14

def area(shape: Shape) = {
    shape match {
        case Circle(r) => pi * r * r
        case Square(x) => x * x
        case Rectangle(x,y) => x * y
    }
}
We Can Define Arbitrary Functions Without Modifying Data Definitions

```scala
def makeLikeFirst(shape0: Shape, shape1: Shape) = {
  (shape0, shape1) match {
    case (Circle(r), Square(s)) => Circle(s)
    case (Circle(r), Rectangle(l,w)) => Circle((l+w)/2)
    case (Square(s), Circle(r)) => Square(r)
    case (Square(s), Rectangle(l,w)) => Square((l+w)/2)
    case (Rectangle(l,w), Circle(r)) => Rectangle(r,r)
    case (Rectangle(l,w), Square(s)) => Rectangle(s,s)
    case _ => shape1
  }
}
```
But A New Data Variant Requires Us To Modify All Functions Over the Datatype

val pi = 3.14

def area(shape: Shape) = {
  shape match {
    case Circle(r) => pi * r * r
    case Square(x) => x * x
    case Rectangle(x,y) => x * y
    case Triangle(b,h) => b*h/2
  }
}


def makeLikeFirst(shape0: Shape, shape1: Shape) = {
  (shape0, shape1) match {
    case (Circle(r), Square(s)) => Circle(s)
    case (Circle(r), Rectangle(l,w)) => Circle((l+w)/2)
    case (Circle(r), Triangle(b,h)) => Circle(b)
    case (Square(s), Circle(r)) => Square(r)
    case (Square(s), Rectangle(l,w)) => Square((l+w)/2)
    case (Square(s), Triangle(b,h)) => Square(b+h/2)
    case (Rectangle(l,w), Circle(r)) => Rectangle(r,r)
    case (Rectangle(l,w), Square(s)) => Rectangle(s,s)
    case (Rectangle(l,w), Triangle(b,h)) => Rectangle(b,h)
    // plus all the cases for Triangle on the left (omitted)
    case _ => shape1
  }
}

But A New Data Variant Requires Us To Modify All Functions Over the Datatype
Recursively Defined Datatypes
Recursively Defined Datatypes

- Case classes allow us to combine multiple pieces of a data into a single object.
- But sometimes we don’t know how many things we wish to combine.
- We can use recursion to define datatypes of unbounded size.
- This case corresponds to the Composite Design Pattern.
Backus-Naur Form
For Lists of Ints

List ::= Empty
       | Cons(Int,List)
Examples of Lists

Empty
Cons(3, Empty)
Cons(3, Cons(1, Empty))
Cons(3, Cons(1, Cons(4, Empty))))
Defining Lists With Scala
Case Classes

abstract class List
case object Empty extends List
case class Cons(head: Int, tail: List) extends List
Where Do We Put Functions Over Lists?

- We do not expect to define new subtypes of lists

- We do expect to define many new functions over lists

- Similar to our Case Two Design Template for Abstract Datatypes

- Thus, we will start with our pattern matching template
An Example Function for Lists

def containsZero(xs: List): Boolean = {
  xs match {
    case Empty => false
    case Cons(n, ys) => {
      if (n == 0) true
      else containsZero(ys)
    }
  }
}
An Example Function for Lists

def containsZero(xs: List): Boolean = {
    xs match {
        case Empty => false
        case Cons(n, ys) => (n == 0) || containsZero(ys)
    }
}
Generalizing to Our First Template Function for Lists

def ourFunction(xs: List): Boolean = {
    xs match {
        case Empty => ...
        case Cons(n, ys) => ...
    }
}
Generalizing to Our First Template Function for Lists

```scala
def ourFunction(xs: List): Boolean = {
  xs match {
    case Empty => ...
    case Cons(n, ys) => ... n ... ourFunction(ys) ...
  }
}

We need to determine our base case
```
Generalizing to Our First Template Function for Lists

def ourFunction(xs: List): Boolean = {
    xs match {
        case Empty => …
        case Cons(n, ys) => … n … ourFunction(ys) …
    }
}

We must determine how to combine these values
def ourFunction(xs: List): Boolean = {
  xs match {
    case Empty => ...
    case Cons(n, ys) => ... n ... ourFunction(ys) ...
  }
}

This template is an example of natural recursion or structural recursion: We recursively decompose and then recombine a computation according to the natural structure of the data.
def containsZero(xs: List): Boolean = {
  xs match {
    case Empty => false
    case Cons(n, ys) => (n == 0) || containsZero(ys)
  }
}

Here the base case is easy:
An empty list does not contain zero (or anything else)
Filling in the Template

```scala
def containsZero(xs: List): Boolean = {
  xs match {
    case Empty => false
    case Cons(n, ys) => (n == 0) || containsZero(ys)
  }
}
```

We break into cases based on the pieces from `match`: Either our first element \( n \) is zero or the answer lies with the rest of the list.
Another Example: How Many Elements?

```scala
def length(xs: List): Int = {
  xs match {
    case Empty => 0
    case Cons(n, ys) => 1 + length(ys)
  }
}
```
Another Example: The Sum of the Elements

def sum(xs: List): Int = {
    xs match {
        case Empty => 0
        case Cons(n, ys) => n + sum(ys)
    }
}
Another Example:
The Product of the Elements

def product(xs: List): Int = {
    xs match {
        case Empty => 1
        case Cons(n, ys) => n * product(ys)
    }
}
Converting Hours to Seconds

**Problem Statement:** Given a list of times measured in hours, we want to construct a list of corresponding times measured in seconds.
Converting Hours to Seconds

```scala
def hoursToSeconds(xs: List): List = {
  xs match {
    case Empty => Empty
    case Cons(n, ys) => Cons(seconds(n), hoursToSeconds(ys))
  }
}

def seconds(hours: Int) = 3600 * hours
```
Generalizing to a Template

def ourFunction(xs: List): List = {
    xs match {
        case Empty => …
        case Cons(n, ys) => Cons(…n…, ourFunction(ys))
    }
}

Really, this is the same template as before, but now Cons is our combining operation
The Natural Numbers

Nat ::= 0
     | Next(Nat)
The Natural Numbers

Nat ::= 0
   | Next(Nat)

Here we are between Cases One and Two for Abstract Datatypes:

- No new variants expected
- Many new functions expected
- But some basic functions are intrinsic to the type
Defining The Natural Numbers in Scala

abstract class Nat

case object Zero extends Nat

case class Next(n: Nat) extends Nat
Defining The Natural Numbers in Scala

abstract class Nat {
    def +(n: Nat): Nat
    def *(n: Nat): Nat
}
Defining The Natural Numbers in Scala

case object Zero extends Nat {
  def +(n: Nat) = n
  def *(n: Nat) = Zero
}

case class Next(n: Nat) extends Nat {
  def +(m: Nat) = Next(n + m)
  def *(m: Nat) = m + (n * m)
}
case object Zero extends Nat {
  def +(n: Nat) = n
  def *(n: Nat) = Zero
}

case class Next(n: Nat) extends Nat {
  def +(m: Nat) = Next(n + m)
  def *(m: Nat) = m + (n * m)
}
Example Reduction

$(3 + 2)$

\[
\begin{align*}
\text{Next(Next(Next(Zero))) + Next(Next(Zero))} & \mapsto \\
\text{Next(Next(Next(Zero)) + Next(Next(Zero)))} & \mapsto \\
\text{Next(Next(Zero) + Next(Next(Zero)))} & \mapsto \\
\text{Next(Next(Zero + Next(Next(Zero))))} & \mapsto \\
\text{Next(Next(Next(Next(Next(Zero))))))} & \\
\end{align*}
\]
def factorial(n: Nat): Nat = {
    n match {
        case Zero => Next(Zero)
        case Next(m) => n * factorial(m)
    }
}
Transferring The Pattern To Ints

```python
def factorial(n: Int): Int = {
    require (n >= 0)

    if (n == 0) 1
    else n * factorial(n - 1)
}
```
Combining Via Auxiliary Functions
Combining Via Auxiliary Functions

• As our examples with natural numbers shows, it is often necessary to define the combining operation of a natural recursion as an auxiliary function

• We can apply this insight to lists and use our template to cover yet more cases
We need to explain how to insert into a sorted list.
def insert(n: Int, xs: List): List = {
  xs match {
    case Empty => Cons(n, Empty)
    case Cons(m, ys) => {
      if (n <= m) Cons(n, xs)
      else Cons(m, insert(n, ys))
    }
  }
}
def insert(n: Int, xs: List): List = {
    xs match {
        case Empty => Cons(n, Empty)
        case Cons(m, ys) => {
            if (n <= m) Cons(n, xs)
            else Cons(m, insert(n, ys))
        }
    }
}
Appending Two Lists

abstract class List {
    /**<
     * Returns a new list with the elements of
     * this list appended to the given list.
     */
    def ++(ys: List): List
}
Appending Two Lists

case object Empty extends List {
    def +(ys: List) = ys
}

Appending Two Lists

case class Cons(first: Int, rest: List) extends List {
  def ++(ys: List) = Cons(first, rest ++ ys)
}

Family Trees

TreeNode ::= Empty
  | Child(TreeNode, TreeNode, Int, String)
abstract class TreeNode

case object EmptyNode extends TreeNode

case class Child(mother: TreeNode, father: TreeNode, yearOfBirth: Int, eyeColor: String) extends TreeNode
Family Trees

def hasBlueEyedAncestor(t: TreeNode): Boolean = {
  t match {
    case EmptyNode => false
    case Child(m,f,b,e) => ((e == "Blue") ||
      hasBlueEyedAncestor(m) ||
      hasBlueEyedAncestor(f))
  }
}
Binary Search Trees
Binary Search Trees

• We will define trees containing only Ints

• To help us find elements quickly, we will abide by the following invariant:

  • At a given node containing value $n$:
    • All values in the left subtree are less than $n$
    • All values in the right subtree are greater than $n$
Binary Search Trees

abstract class BinarySearchTree {
    def contains(n: Int): Boolean
    def insert(n: Int): BinarySearchTree
}

case object EmptyTree extends BinarySearchTree {
   def contains(n: Int) = false
   def insert(n: Int) = ConsTree(n, EmptyTree, EmptyTree)
}
case class ConsTree(m: Int,
    left: BinarySearchTree,
    right: BinarySearchTree)
extends BinarySearchTree {

  def contains(n: Int): Boolean = {
    if (n < m) left.contains(n)
    else if (n > m) right.contains(n)
    else true // n == m
  }

  def insert(n: Int) = {
    if (n < m) ConsTree(m, left.insert(n), right)
    else if (n > m) ConsTree(m, left, right.insert(n))
    else this // n == m
  }
}
What if we call **insert** with 143?
What if we call `insert` with 143?